NASA Contractor Report 189625
ICASE Report No. 92-12

ICASE

THE DESIGN AND IMPLEMENTATION OF A PARALLEL UNSTRUCTURED EULER SOLVER USING SOFTWARE PRIMITIVES

R. Das
D. J. Mavriplis
J. Saltz
S. Gupta
R. Ponnusamy

Contract No. NAS1-18605
March 1992

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

THE DESIGN AND IMPLEMENTATION OF A PARALLEL UNSTRUCTURED EULER SOLVER USING SOFTWARE PRIMITIVES

R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, R. Ponnusamy[1]
ICASE
NASA Langley Research Center
Hampton, VA 23665

ABSTRACT

This paper is concerned with the implementation of a three-dimensional unstructured-grid Euler-solver on massively parallel distributed-memory computer architectures. The goal is to minimize solution time by achieving high computational rates with a numerically efficient algorithm. An unstructured multigrid algorithm with an edge-based data-structure has been adopted, and a number of optimizations have been devised and implemented in order to accelerate the parallel computational rates. The implementation is carried out by creating a set of software tools, which provide an interface between the parallelization issues and the sequential code, while providing a basis for future automatic run-time compilation support. Large practical unstructured grid problems are solved on the Intel iPSC/860 hypercube and Intel Touchstone Delta machine. The quantitative effect of the various optimizations are demonstrated, and we show that the combined effect of these optimizations leads to roughly a factor of three performance improvement. The overall solution efficiency is compared with that obtained on the CRAY-YMP vector supercomputer.

[1]Research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

1. INTRODUCTION

The relatively rapid growth in microprocessor technology over the last decade has lead to the development of massively parallel hardware capable of solving large computational problems. Given the relatively slow increases in large mainframe supercomputer capabilities, it is now generally acknowledged that the most feasible and economical means of solving extremely large computational problems in the future will be with highly parallel distributed memory architectures. While such hardware already exists and is rapidly progressing, the software required to solve large problems in parallel has proved to be a major stumbling block. The software problems are twofold. First, an efficient and inherently parallelizable algorithm must be devised for solving the problem at hand. Algorithmic efficiency relates to the ability to solve a problem with a minimum number of operations or iterations. Thus, simple explicit schemes are not generally suitable for large problems, and more complex implicit or multigrid schemes which propagate information more rapidly throughout the domain are preferred. A parallelizable algorithm is one which can be broken up into smaller components and executed on a parallel architecture without incurring substantial overheads, both in terms of reduced computational efficiency (increased number of operations to achieve the same level of convergence) and increased communication costs. While simple explicit schemes are often easily parallelizable, they are not efficient. On the other hand, considerable difficulty has often been experienced in parallelizing more complex algorithms. This problem is compounded in the case of unstructured grids, since relatively few efficient algorithms have been devised, even for sequential (vector) architectures. The second software problem is an implementation issue. Most often, the programmer is required to explicitly distribute large arrays over multiple local processor memories, and keep track of which portions of each array reside on which processors. In order to access a given element of a distributed array, communication between appropriate processors must be invoked. The programmer should be relieved of such machine dependent and architecture specific tasks. Ultimately, a parallelizing compiler should be capable of automatically distributing data and setting up inter-processor communication in an efficient manner, much as present-day coarse grained shared memory supercomputers provide automated support for multi-processing. Such low-level implementational issues have severely limited the growth of parallel computational applications, much in the same way as vectorized applications were inhibited early on, prior to the development of efficient vectorizing compilers.

This work represents the combination of two related efforts aimed at easing the software problem. On the one hand, an efficient three-dimensional unstructured solver has been developed which is highly parallelizable. The sequential version of this algorithm has previously been reported [1]. The data-structures and solution strategy (multigrid) have been designed

(or chosen) with parallel overhead issues in mind. (These also have a beneficial effect on the sequential code). On the other hand, a set of parallelization software tools (primitives) has been developed in order to ease the task of implementing the present code on parallel architectures. The development of a parallelizing compiler is planned which eventually will be capable of automatically imbedding the primitives in the source code. However, at this stage, the user is required to explicitly invoke the primitives at the appropriate locations in the source code. While this does not provide a completely automatic parallelization tool, it does relieve the user of the most burdensome low-level details of the implementation. The development of these primitives, known as the PARTI primitives (Parallel Automated Runtime Toolkit at ICASE), as well as the compiler, has been underway for some time, and has been previously reported [2]. This paper is less concerned with the individual development of the solver or the parallelizing primitives, but more so with the interaction between these two efforts resulting from a specific application: the implementation of the unstructured Euler solver on the Intel iPSC860. For example, the edge-based data-structure employed in the solver was originally chosen in order to minimize and simplify communication between processors. Similarly, experience gained during the implementation was used to modify various primitives and even to create new primitives whose functionality had not been foreseen.

Parallel unstructured solver implementations have been performed in two-dimensions by various authors, on SIMD architectures [3], and on distributed memory MIMD architectures [4], [5] and [6]. Software environments for irregular problems have also been developed and applied to two-dimensional unstructured grid solvers [5], and to two dimensional unstructured multigrid solvers While much use has been made of the concepts developed in previous work, new optimizations have also been devised and incorporated. This collection of techniques has been encapsulated into a set of software primitives which provides the interface between the parallel implementation and the sequential code.

In the next section, a brief description of the unstructured multigrid Euler solver is given. Thereafter, a description of the general parallelization approach and of the various optimizations is given, as well as a description of the underlying philosophy of software-tool development. The goal of the results section is to illustrate the beneficial effects on performance of each of the individual optimizations, and to compare the obtained performance with that of present-day supercomputers. Finally, the solution of a large practical unstructured grid problem is demonstrated and the parallel efficiency is compared with that obtained on the CRAY-YMP-8 vector supercomputer.

2. THREE DIMENSIONAL MULTIGRID EULER SOLVER

The basis for the implementation is a three dimensional unstructured mesh Euler solver.

Unstructured meshes provide a great deal of flexibility in discretizing complex domains and offer the possibility of easily performing adaptive meshing. However, unstructured meshes result in random data-sets and large sparse matrices, which pose a significant challenge for parallelization issues.

The three dimensional compressible gas dynamics equations are discretized on the unstructured mesh using a Galerkin finite-element approach [1], [7]. The flow variables are stored at the vertices of the mesh, and piecewise linear flux functions are assumed over the individual tetrahedra of the mesh. The resulting spatially discretized equations can be recast as a summation at each vertex of contributions along all edges meeting at that vertex. Thus, the convective residuals may be assembled by performing a simple loop over the edges of the mesh. Artificial dissipation terms are required in order to stabilize the solution, and these are constructed as a blend of a Laplacian and biharmonic operator, the former being constructed as a single loop over edges, and the latter as a double edge-loop. The spatially discretized equations thus form a large set of coupled ordinary differential equations, which must be integrated in time to obtain the steady-state solution. This is achieved using a 5-stage Runge-Kutta scheme. Enthalpy damping, local time-stepping, residual averaging and an unstructured multigrid algorithm are employed to accelerate convergence to steady-state. In the multigrid algorithm, at each cycle a single time-step is first performed on the finest grid of the sequence and the flow variables and residuals are then interpolated up to a coarser grid. This process continues recursively, on successively coarser grids. When the coarsest grid is reached, the corrections are interpolated back to each successively finer grid, and a new cycle is initiated. In the context of unstructured meshes, it has proven useful to rely on sequences of independent non-nested coarse and fine meshes. In order to efficiently interpolate variables between such meshes, an efficient search algorithm must be invoked in order to determine the patterns (addresses and weights) for interpolation between any two consecutive meshes of the sequence. This is done in a preprocessing stage, on a sequential machine, prior to the flow computations. Alternatively, this may be viewed as a mesh generation post-processing stage. The basic data-structure for the Euler solver is based on the mesh edges. For each edge, we store the addresses of the two vertices on either end of the edge (similar to the coordinate-storage scheme for sparse matrices). This represents the minimum amount of information necessary to describe the unstructured grid. It also results in the minimum amount of data transfer between adjacent vertices within a residual evaluation operation, by avoiding duplicate transfers which are usually incurred by face-based and traditional finite-element cell based data-structures. For parallel implementations, this results in the minimum amount of communication and enables a relatively simple implementation, since each edge can be shared by at most two processors. The multigrid interpolation

procedures, in which the addresses and weights for interpolation have been precomputed, can be viewed as a simple gather/scatter of data from one array (grid) to another. Such operations are similar to the gather-scatter operations required on a given grid for assembling the residuals and can therefore be implemented with existing software primitives.

3. PARALLELIZATION (PARTI) PRIMITIVES

The PARTI primitives (Parallel Automated Runtime Toolkit at ICASE) are designed to ease the implementation of computational problems on parallel architecture machines by relieving the user of the low-level machine specific issues. The research described in this paper began with the version of PARTI described in [8] and surveyed in this section. We then proceeded to identify ways in which the performance of unstructured codes could be optimized. The optimizations that involved reduction of communication overheads resulted in an improved version of PARTI. The optimizations that involved reduction of computation time were manually implemented.

The PARTI primitives enable the distribution and retrieval of globally indexed but irregularly distributed data-sets over the numerous local processor memories. In distributed memory machines, large data arrays need to be partitioned between local memories of processors. These partitioned data arrays are called distributed arrays. Long term storage of distributed array data is assigned to specific memory locations in the distributed machine. A processor that needs to read an array element must fetch a copy of that element from the memory of the processor in which that array element is stored. Alternately, a processor may need to store a value in an off-processor distributed array element. Thus, each element in a distributed array is assigned to a particular processor, and in order to be able to access a given element of the array we must know the processor in which it resides, and its local address in this processor's memory. We thus build a translation table which, for each array element, lists the host processor address. For a one-dimensional array of N elements, the translation table also contains N elements, and therefore must be distributed itself over the local memories of the processors. This is accomplished by putting the first N/P elements on the first processor, the second N/P elements on the second processor, etc ..., where P is the number of processors. Thus, if we are required to access the m^{th} element of the array, we look up its address in the distributed translation table, which we know can be found in the $(m/P + 1)^{th}$ processor. Alternatively, we could simply renumber all the vertices of the unstructured grid in order to obtain a regular partitioning of arrays over the processors. However, the present approach can easily deal with arbitrary partitions, and should enable a straight-forward implementation of dynamically varying partitions, which may be encountered in the context of adaptive meshing. One of the primitives handles initializa-

tion of distributed translation tables, and another primitive is used to access the distributed translation tables.

PARTI carries out optimizations which reduce both the number of messages sent as well as the volume of data that must be communicated. In distributed memory MIMD architectures, there is typically a non-trivial communications latency or startup cost. For efficiency reasons, information to be transmitted should be collected into relatively large messages. The cost of fetching array elements can be reduced by precomputing what data each processor needs to send and to receive. In irregular problems, such as solving PDEs on unstructured meshes and sparse matrix algorithms, the communications pattern depends on the input data. This typically arises due to some level of indirection in the code. This lack of information is dealt with by transforming the original parallel loop into two constructs called an *inspector* and *executor*. During program execution, the inspector examines the data references made by a processor, and calculates what off-processor data needs to be fetched and where that data will be stored once it is received. The executor loop then uses the information from the inspector to implement the actual computation. The PARTI primitives can be used directly by programmers to generate inspector/executor pairs. Each inspector produces a communications *schedule*, which is essentially a pattern of communication for exchanging data.

Significant work has gone into optimizing the gather, scatter and accumulation communication routines for the iPSC/860. During the course of developing the PARTI primitives, we experimented with a large number of ways of writing the kernels of our communication routines. It is not the purpose of this paper to describe these low level optimizations or their effects in detail; we will just summarize the best communication mechanism we have found. *In all of the experimental studies reported in this paper, we use the optimized version of the communication routine kernels.*

We communicate using FORCED message types. We use non-blocking receive calls (irecv), each processor posts all receive calls befcre it sends any data. Synchronization messages are employed to make sure that an appropriate receive has been posted before the relevant message is sent.

Communications contention is also reduced. We use a heuristic developed by Venkatakrishnan [4] to determine the order in which each processor sends out its messages. The motivation for this heuristic is to reduce contention by dividing the communication into groups of messages such that within each group, each processor sends and receives at most one message. As Venkatakrishnan points out, this heuristic makes the tacit assumption that all messages are of equal length and in any event does not even attempt to eliminate link contention.

4. COMMUNICATIONS OPTIMIZATIONS

Our communication optimizations reduce the quantity of data that must be transmitted between processors, the optimizations also reduce the number of messages that must be sent.

In our unstructured mesh solver, we encounter a variety of situations in which the same data is accessed by several consecutive loops. For instance, consider a step of the Runge Kutta integration. Flow variables are used in sequence of three loops over edges followed by a loop over boundary faces. The flow variables are only updated at the end of each of the Runge Kutta steps. We can obtain all of the off-processor flow variables needed at the beginning of the step. This makes it advantageous to develop methods that avoid bringing in the same data more than once, these methods can also reduce the number of communication startups.

Our new methods make it possible to track and reuse off-processor data copies. We do this by modifying our software so that we are able to generate *incremental* communications schedules. Incremental schedules obtain only those off-processor data not requested by a given set of pre-existing schedules. The pictorial representation of an *incremental schedule* is given in Figure 1. In this figure, depict a situation in which a loop over mesh edges (edge_loop) is followed by a loop over mesh boundary faces (face_loop). The schedule to bring in the off-processor data for the edge_loop is given by the *edge schedule* and is formed first. During the formation of the schedule to bring in the off-processor data for the face_loop we remove the duplicates shown by the shaded region in Figure 1. Removal of duplicates is achieved by using a hash table. The off-processor data to be accessed by the edge schedule is first hashed using a simple hash function. Next all the data to be accessed during the face_loop is hashed. At this point the information that exists in the hash table allows us to remove all the duplicates and form the *incremental schedule.*

5. REORDERING COMPUTATION

The performance of a single i860 processor on computationally important loops over mesh edges range from 2.75 MFLOPS to 4.1 MFLOPS, in 64 bit arithmetic, depending on the unstructured mesh used. It seems likely that this relatively poor performance is due to the effects of irregular data access patterns on the i860 memory hierarchy. The lowest level of i860 memory hierarchy consists of 32 integer and 32 floating point registers, each 32-bits wide. Data stored in registers can be used directly for computation without any memory overheads. At the next level of memory hierarchy, i860 has 8k byte data cache. The cache line size is 128 bits and two double precision floating point numbers can be loaded from main memory into cache simultaneously. The data stored in cache can be accessed with a delay of

one clock cycle. In cases where data is not available in cache it is loaded from memory, which causes a delay of several cycles. Other than possibly poor utilization of registers and cache, highly irregular data access patterns can potentially cause severe performance degradation due to overheads associated with the i860's mechanism for handling virtual memory. The virtual memory of i860 is divided into 4k bytes pages. Whenever a new page is accessed, a substantial overhead is incurred, associated with locating the page in physical memory. This overhead can be saved if the page has recently been accessed. The details of each component of i860 memory hierarchy can be found in [9].

The single processor performance can be improved by reordering the data and by restructuring the code associated with unstructured mesh computations. Because we employ an edge based data structure in our codes, most of the computational work in the code is found in loops over edges so we concentrate our efforts on such loops. We investigated methods that change the order in which mesh edges are traversed in loops. We also investigated methods which renumber mesh vertices and reorder data associated with the mesh. Reordering edges and renumbering vertices is expected to result in better locality and therefore should improve cache utilization and reduce virtual memory management overheads. We also investigated the effects of restructuring edge loops to use a compressed sparse row (CSR) type representation to improve register utilization and reduce the number of memory operations.

Reordering Edges

We reorder list of edges so that all the edges incident on a node are listed consecutively. The edges incident on node 1 are listed first, followed by edges incident on node 2 and so on. We avoid listing and edge (i, j) twice by associating it with node i, if $i < j$, or with node j if $j < i$. The advantage of this reordering is that for all edges associated with a node i, the data for node i remains in cache, giving a better cache utilization.

Loop Restructuring

We can restructure edge loops to use a CSR type format [10]. When we reorder edges, we list all edges incident on a node consecutively. We can take this a step further by compressing the data structure that represents the list of edges. An edge represents an association between two vertices. Once we have ordered the edges in the manner described above, for each edge (i, j) we can simply generate an array **IA** that lists the consecutive values of j. We maintain a separate pointer array **JA**. Thus IA(JA(i)),..,IA(JA(i+1)-1) represent the edges associated with node i.

Node Reordering

The numbering of mesh vertices can have an important effect on the pattern of data access. We seek to number mesh vertices so that data associated with vertices linked by mesh edges tend to be stored in nearby memory locations. There is no reason to expect that mesh orderings produced by mesh generators should have this property. For instance, Figure 2 depicts data access pattern for the 2800 node mesh shown in Figure 4. The X-axis and the Y-axis in the figure give node numbers and each point represents an edge between the corresponding vertices on X-axis and Y-axis.

We expect that the highly non-uniform data accesses shown in Figure 2 will cause poor cache utilization and high virtual memory management overheads. To reduce these overheads, we reorder the vertices using the Reverse Cuthill McKee (RCM) method [11]. The RCM reordering method is frequently used by researchers in the area of sparse matrix computation. It is a graph based technique to reorder columns of a sparse matrix to reduce its bandwidth. The resulting data access pattern after RCM reordering in shown in Figure 3.

In comparison to Figure 2, the data access pattern of Figure 3 shows much less irregularity and therefore, should improve cache utilization and reduce virtual memory management overheads.

Single Processor Results

The above reordering methods were applied to several meshes to study single node performance on the iPSC/860. The results of our experimentation are summarized in Table 1. Meshes M1 and M2 are "regular" meshes which are represented using our unstructured mesh data structures (i.e. tetrahedral meshes derived from the subdivision of a structured hexahedral mesh), and mesh M3 is the smallest of the unstructured meshes used by the multigrid algorithms to solve the transonic test case over the ONERA M6 wing.

Column one in the above table gives the number of vertices and identification of the mesh used. The MFLOPS obtained without any reordering by the unstructured Euler solver are shown in Column 2. The next four columns show the improvement in MFLOPS due to various combinations of three reordering schemes described above. The results indicate that edge reordering alone gives significant improvement in performance for all the meshes. Node reordering in conjunction with edge reordering has more impact on the mesh M3 in comparison to the two other meshes. This is probably due to the differences in the ways in which the meshes were generated. The loop transformation does not have much impact on performance (Column 3 v/s Column 5 and Column 4 v/s Column 6) because use of cache instead of registers is not very costly. From these experimental results we conclude that

by reordering the performance on a single node of the iPSC/860 can be improved almost by a factor of two for the types of meshes used in unstructured applications. Since loop restructuring was found to yield only marginal benefits, in the interest of preserving the original structure of the sequential code, only edge reordering and node reordering were employed in all subsequent implementations.

6. RESULTS

We describe the results of a number of experiments we have carried out to evaluate the performance impact of our optimizations. These experiments were carried out on an Intel iPSC/860 hypercube and the Intel Touchstone Delta. For purposes of comparison, we cite performance numbers obtained from an optimized Cray YMP version of this code [1].

A standard transonic test case is chosen for this comparison, namely a Mach 0.84 flow over an ONERA M6 wing at 3.06 degrees incidence. The sequence of meshes employed for the multigrid algorithm in this case are depicted in Figure 4. The coarsest grid contains merely 2,800 vertices, while the finest grid contains a total of 357,900 vertices and just over 2 million tetrahedra. The intermediate grids have 9,428 vertices and 53,961 vertices respectively. The computed Mach contours of the obtained solution are depicted in Figure 5, where the familiar double-shock pattern is observed.

We employ a single grid algorithm along with two versions of multigrid algorithms. The two versions of multigrid are W and V-cycle algorithms. The V-cycle multigrid algorithm visits all meshes an equal number of times within a single cycle, while the W-cycle visits coarse meshes more frequently than fine meshes. Details of these algorithms can be found in [1]. W-cycle multigrid strategies require slightly more work per cycle (of the order of 15% to 25% depending on mesh sizes), but often converge slightly more rapidly and are thus more efficient overall. However, the relative merits of W versus V-cycle strategies can be very case dependent. On the other hand, both strategies always offer large increases in efficiency over single grid explicit methods. Figure 6 provides a performance comparison between the single grid and W-cycle multigrid code by plotting convergence histories in terms of work units for the solution of flow over the ONERA M6 wing at the above prescribed conditions on the 357K mesh. In this plot, a work unit is defined as the time required for a single grid explicit cycle. As can be seen in this figure, the W-cycle multigrid algorithm converges over 6 orders of magnitude in roughly 150 work units, which corresponds to 100 multigrid cycles, while the single grid calculation is seen to require almost an order of magnitude more work to reach the same level of convergence.

On the CRAY-YMP, the single-grid code for this case required a total of 33MW of memory and ran at a speed of 19 seconds/cycle on a single processor. The W-cycle multigrid

code required a total of 42 MW of memory and ran at a speed of 34 seconds/multigrid cycle. For both cases, the computational rates achieved were about 100 Mflops. For the multigrid run, engineering solutions (3 to 4 orders of convergence) for this case could thus be obtained in roughly 30 minutes of CRAY-YMP single processor CPU time.

We employed the recursive spectral partitioning algorithm to carry out partitioning [12], [13]. Williams [14] compared this algorithm with binary dissection [15] and simulated annealing methods for partitioning two dimensional unstructured mesh calculations. He found that recursive spectral partitioning produced better partitions than binary dissection. Simulated annealing in some cases produced better partitions but the overhead for simulated annealing proved to be prohibitive even for the relatively small meshes employed (the largest had 5772 elements). Venkatakrishnan [4] and Simon [13] also reported favorable results with this partitioner. We carried out preliminary performance comparisons between binary dissection and the recursive spectral partitioning and found that recursive spectral partitioning gave superior results on the iPSC/860 on our three dimensional meshes. The results we report all have been obtained using recursive spectral partitioning to partition all meshes. Partitioning was performed on a sequential machine as a preprocessing operation. In all of the experimental studies reported in this paper, we use the same optimized version of the communications kernels which employed forced message types, non-blocking receives (irecv), and which employ Venkatakrishnan's heuristic to determine the order in which messages are sent.

Table 2 examines the effects of the communication optimizations and the reordering optimizations. The table depicts the Mflops obtained with the single grid, and the W and V cycle multigrid algorithms on a 357K node fine mesh. The figure also depicts the performance of the single grid method on the second finest grid (53,961 vertices). Measurements were performed on a 128 processor Intel iPSC/860. We achieve roughly a factor of 2.5 to 3 improvement in computational rate from the use of our optimizations. For instance, consider the single mesh code on the 357K mesh. When we employed both optimizations, we saw a computational rate of 356 Mflops. Communications optimizations without reordering yielded 217 Mflops, reordering without communications optimizations yielded 149 Mflops. When we employed neither optimization, we achieved 127Mflops. Analogous improvements are seen in the multigrid codes.

The multigrid V and W cycle algorithms achieved 298 and 244 Mflops respectively when we employed both communication optimizations and reordering. The frequent visits to coarse meshes, interpolation and prolongation in W cycle multigrid might be expected to lead to a significant degradation in computational rate. A degradation of roughly 32% compared to the single grid code was in fact observed, but substantially larger degradations are seen

when we leave out either our communication optimizations or reordering.

Table 3 gives information on the effects of the different optimizations on communication and computation time. For the single mesh code, use of the communications optimizations lead to a fourfold reduction in time spent on communication. For the W cycle multigrid, we have broken down communication time into

> communication time required to carry out calculations on one of the four individual meshes in the multigrid calculation (intra-mesh communication in Table 3), and
>
> communication time required for transferring data between meshes (inter-mesh communication Table 3).

We note that the cost of carrying out intra-mesh communication is over an order of magnitude higher than the cost of carrying out inter-mesh communication. This indicates that we do not appear to be making a significant performance compromise by independently partitioning the meshes in the multigrid algorithms.

The final test case involves the computation of a highly resolved flow over a three-dimensional aircraft configuration. We employed the single mesh code for this test case, although plan soon to run the multigrid case. The mesh contains a total of 804,056 points and approximately 4.5 million tetrahedra. This is believed to be the largest unstructured grid problem attempted to date. In Figure 7, we depict the second mesh of the proposed multigrid sequence (we do not show the 804K mesh due to printing and resolution limitations). For this case, the freestream Mach number is 0.768 and the incidence is 1.16 degrees. The computed Mach contours are shown in Figure 8, where good resolution of the shock on the wing is observed. This case achieved a rate of 778 Mflops on 256 processors of the Delta machine, and 1496 Mflops on the full 512 processor configuration of the Delta. The same case was run on the CRAY-YMP-8 machine, using all eight processors in dedicated mode. The CRAY autotasking software was used to parallelize the code for this architecture. Both the single grid and multigrid codes achieved a computational rate of 750 Mflops on all eight processors, which corresponds to a speedup of roughly 7.5 over the single processor performance. A residual reduction of six orders of magnitude was obtained in 100 multigrid W-cycles which required 16 minutes on the full CRAY-YMP-8. Although the computational rate achieved on 512 processors of the Delta machine is roughly double of that delivered by the CRAY-YMP-8, a degradation of about 30% can be expected with the implementation of the multigrid W-cycle on the Delta for this case, as was observed for the previous case. Thus, a similar solution should be achievable on the Delta in just under 10 minutes.

In Table 4 we depict the computational rates achieved on different architectures for the single mesh solution procedure, along with V and W cycle multigrid solution procedures,

for the two cases previously described. We employed all of our optimizations in these tests. As expected, for the CRAY-YMP, single processor rates are relatively insensitive to the problem size and the solution strategy. On the iPSC/860 and the Delta the computational rates are calculated by counting the number of floating point operations performed. On the CRAY-YMP, the system facility called *hpm* was utilized to measure the rates. If we scale the computational rate given by the CRAY-YMP facility, to get the rate for the Intel machine, the scaling factor being the ratio of the time per cycle on the CRAY-YMP to the corresponding value on the Intel machine, we find the rate is about an average of 40 Mflops more than that presented for the iPSC/860 and the Delta. On the iPSC/860 and the Delta architectures, maximum computational efficiency is achieved for large meshes using the single grid solution strategy. Thus the single grid run of the 804K grid on 512 Delta processors achieves twice the computational rate of the CRAY-YMP-8, or 15 times the rate of a single CRAY-YMP processor. Similarly, the 357K mesh achieved about 7 times the performance of a single CRAY-YMP processor for a single grid run, (roughly equivalent to the full CRAY-YMP-8 performance) and 4 times the single processor CRAY-YMP performance for a W-cycle multigrid run (or about 60% of the CRAY-YMP-8 performance). Since the overall solution efficiency of the multigrid strategy is much higher than that of the single grid explicit scheme, this emphasizes the need to use overall solution time as a measure of solution efficiency rather than simply computational rates.

7. CONCLUSIONS

A number of earlier reports have noted that two dimensional, explicit unstructured mesh solvers appear to be well suited for distributed memory multiprocessors. While explicit schemes may be easily parallelizable, they are not numerically efficient. We have developed a distributed memory version of an efficient three dimensional unstructured multigrid code. We have shown that competitive computational rates can be achieved for this problem on massively parallel distributed memory architectures. An approximately three-fold performance improvement was obtained by optimizations which reduced communication overhead and increased the computation time required by the processors.

The encapsulation of the communications optimizations into a set of software primitives eases the implementation of similar problems and the porting to different architectures, while providing the foundations for possible run-time compilation support of parallelization [8]. The simultaneous use of an efficient multigrid algorithm and massive parallelism results in rapid solution times for large problems.

Our approach is designed to facilitate the development of parallelized adaptive meshing strategies. The PARTI primitives currently support problem remapping but do not as yet

support the functionality that will be required for the process of adaptive remeshing itself. Once this functionality is developed, we will be able to implement adaptive three dimensional multigrid codes on distributed architectures.

ACKNOWLEDGEMENT

We would like to thank Horst Simon for providing us with his recursive spectral partitioner and Rob Vermeland and CRAY Research Inc. for providing dedicated time on the CRAY-YMP-8 machine. We would also like to acknowledge National Institutes of Health and NAS at NASA Ames for allowing us access to their 128 processor iPSC/860. This research was performed in part using the Intel Touchstone Delta system operated by Caltech on behalf of the Concurrent Supercomputer Consortium. Access to this facility was provided by the National Aeronautics and Space Administration. We would like to thank Bob Voigt, Yousuff Hussaini and Manuel Salas for their encouragement and support over the course of this project.

References

[1] D. J. Mavriplis. Three dimensional multigrid for the euler equations. *AIAA paper 91-1549CP*, pages 824–831, June 1991.

[2] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory architectures. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[3] S. Hammond and T. Barth. An optimal massively parallel euler solver for unstructured grids. *AIAA Journal, AIAA Paper 91-0441*, January 1991.

[4] V. Venkatakrishnan, H. D. Simon, and T. J. Barth. A MIMD implementation of a parallel euler solver for unstructured grids, submitted to Journal of Supercomputing. Report RNR-91-024, NAS Systems Division, NASA Ames Research Center, Sept 1991.

[5] R. Williams. DIME - Distributed irregular mesh environment: User's manual. Report C3P 861, Cal Tech, Feb 1990.

[6] J. De Keyser and D. Roose. Adaptive irregular multiple grids on a distributed memory multiprocessor. In A. Bode, editor, *Proceedings of the 2nd European Distributed Memory Computing Conference*, pages 153–162, 1991.

[7] A. Jameson, T. J. Baker, and N. P. Weatherhill. Calculation of inviscid transonic flow over a complete aircraft. *AIAA paper 86-0103*, January 1986.

[8] J. Saltz, H. Berryman, and J. Wu Runtime compilation for multiprocessors. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

[9] N. Margulis. *i860 Microprocessor Architecture.* McGraw Hill, 1990.

[10] Y. Saad. Sparsekit: a basic tool kit for sparse matrix computations. Report 90-20, RIACS, 1990.

[11] E. H. Cuthill and J. Mckee. Reducing bandwidth of sparse symmetric matrices. In *Proc. ACM 24th National Conference*, pages 157–172, New York, 1969. ACM, New York.

[12] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Mat. Anal. Appl.*, 11:430–452, 1990.

[13] H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications.* Permagon Press, 1991.

[14] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency, Practice and Experience*, 3(5):457–482, February 1991.

[15] M.J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. on Computers*, C-36(5):570–580, May 1987.

Mesh size	Original	Edge Reordering		Loop Restructuring	
		Original Node Ordering	Node reordering	Original Node Ordering	Node reordering
545(M1)	4.03	5.90	6.15	6.53	6.74
3681(M2)	4.09	6.13	6.27	6.42	6.74
2800(M3)	2.76	4.28	5.42	4.35	5.76

Table 1: Performance of reordering on single node of iPSC/860 in Mflops

Method	Explicit 53K Mesh	Explicit 357K Mesh	Multigrid 357K Mesh V Cycle	Multigrid 357K Mesh W Cycle
No Communication Opt. Original Mesh Ordering	85	127	105	92
No Communication Opt. Reordered Mesh	88	149	120	100
Communication Opt. Original Mesh Ordering	172	217	181	153
Communication Opt. Reordered Mesh	216	356	298	244

Table 2: Performance in Mflops of Unstructured Mesh Code on 128 Processor iPSC/860

Method	Explicit		Multigrid W Cycle		
	comm	total	comm		total
			intra-mesh	inter-mesh	
No Communication Opt. Original Mesh Ordering	7.6	14.1	30.1	1.7	40.1
No Communication Opt. Reordered Mesh	7.6	12.0	29.4	1.6	35.8
Communication Opt. Original Mesh Ordering	1.8	8.2	9.6	0.5	22.7
Communication Opt. Reordered Mesh	1.8	5.0	9.4	0.4	14.6

Table 3: Communication, Total Time per Cycle (seconds). Unstructured Mesh Code 128 Processor iPSC/860 357K Mesh

Method	*Explicit* 357K Mesh	*Multigrid* 357K Mesh V Cycle	*Multigrid* 357K Mesh W Cycle
128 Processor iPSC/860	356	298	244
128 Processor Delta	408	320	267
256 processor Delta	646	516	412
1 Processor Y/MP	103	100	100

Table 4: Computational Rates (Mflops) Unstructured Mesh Code iPSC/860 and Delta. Incremental Scheduling, Blocked and Reordered Mesh

INCREMENTAL SCHEDULE

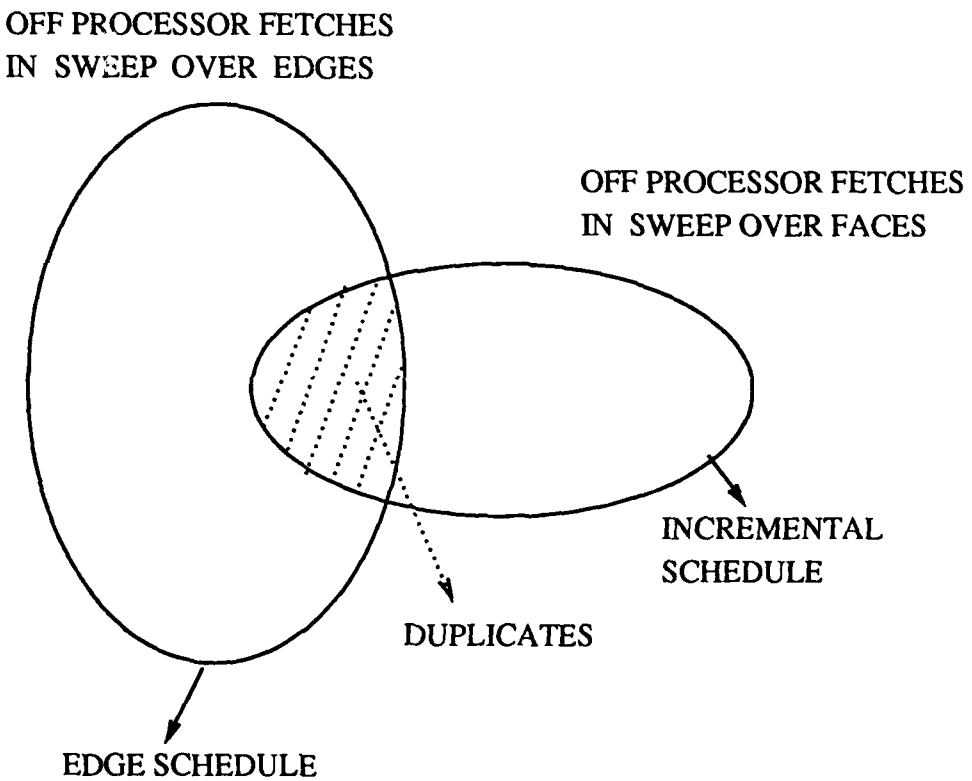


Figure 1: Incremental Schedule

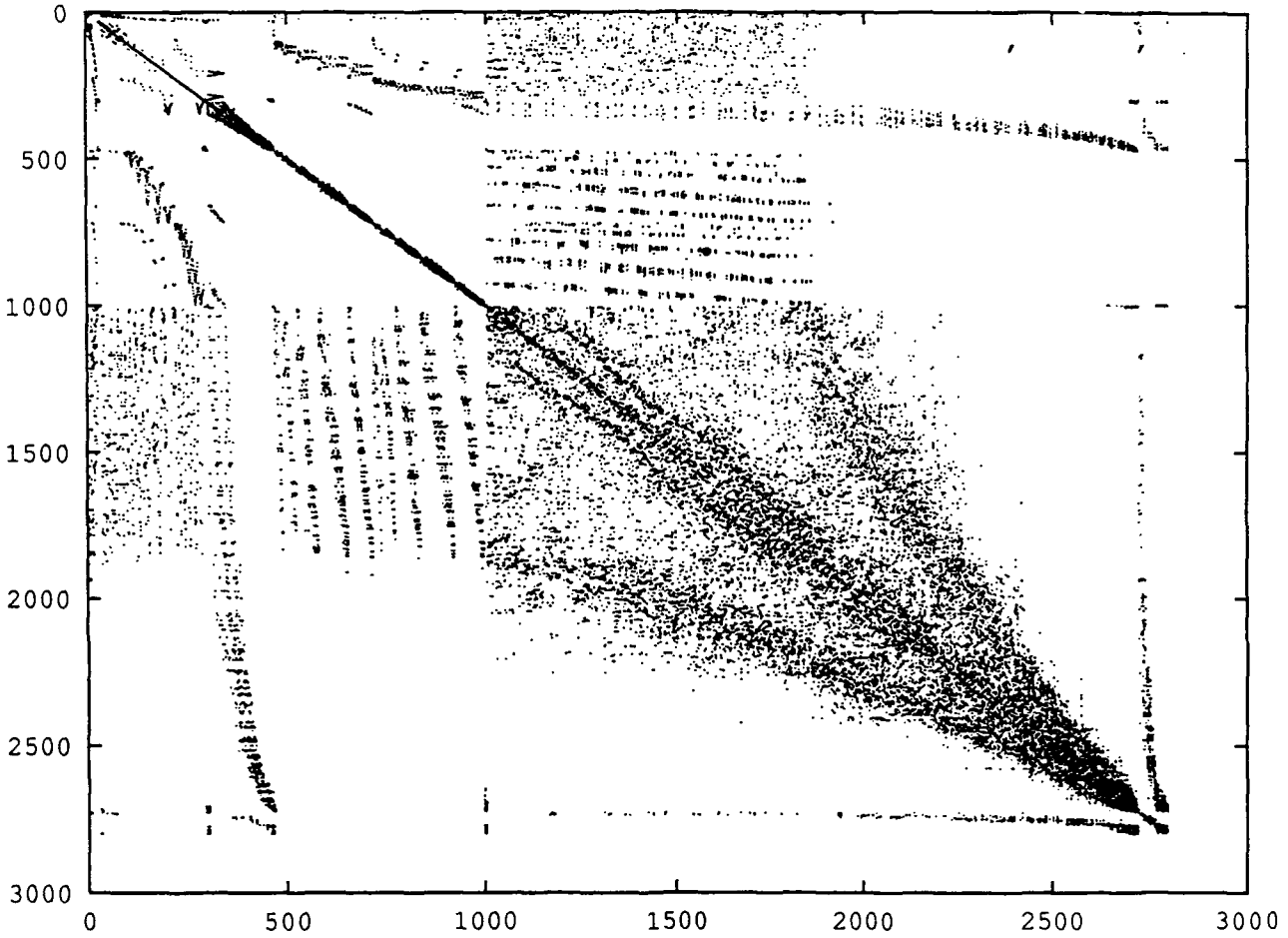


Figure 2: Initial Data Access Pattern of 2800 Node Mesh

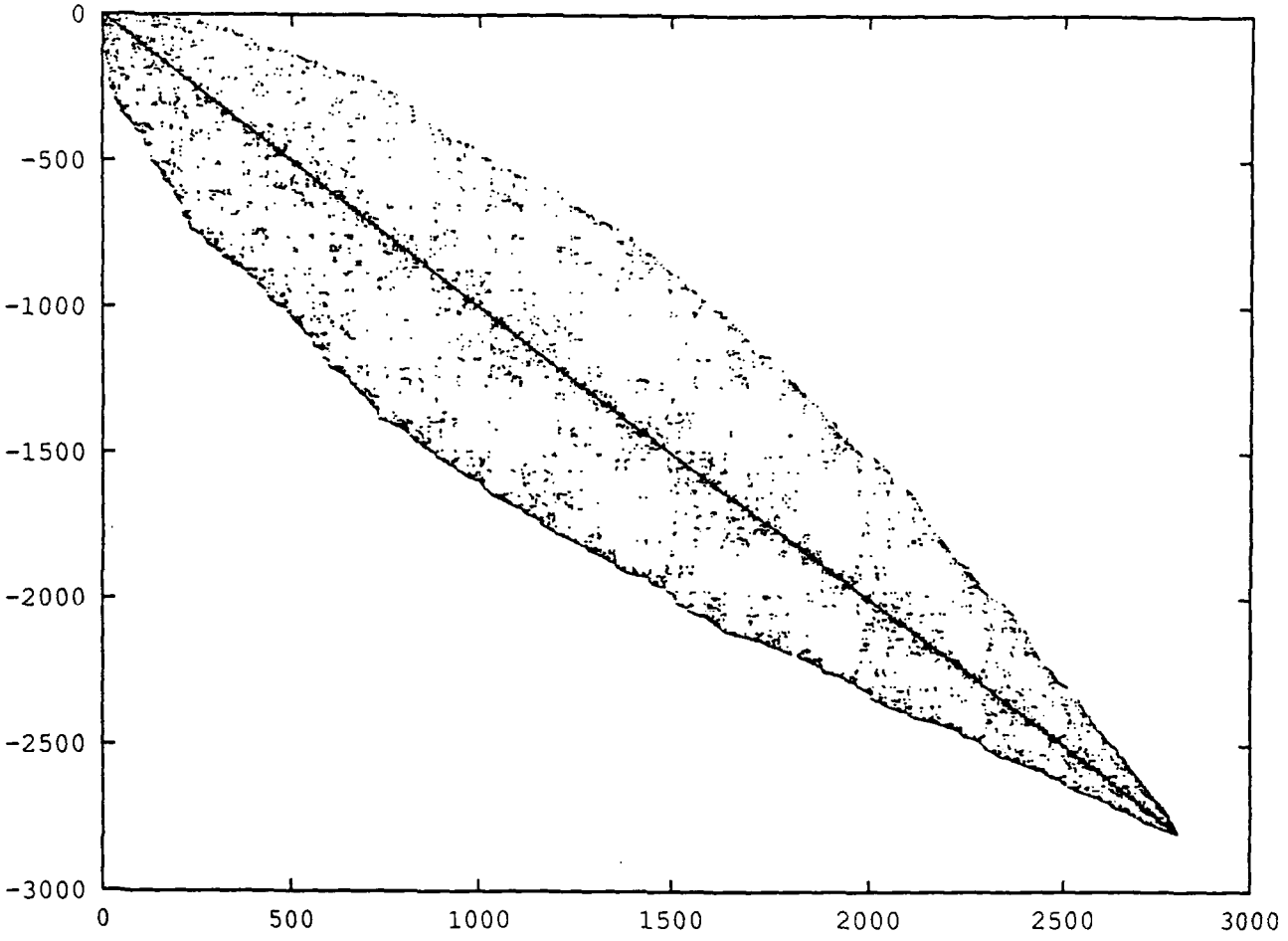
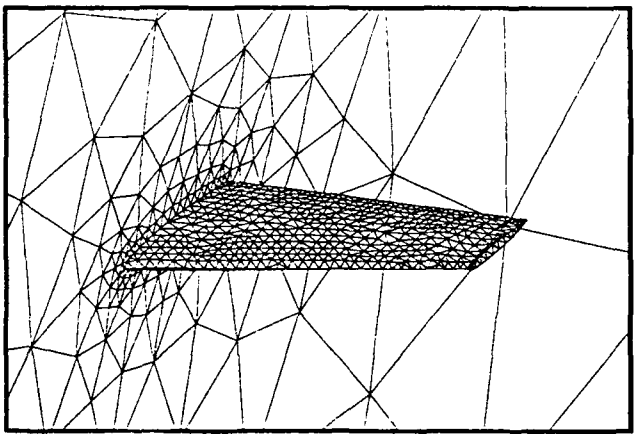
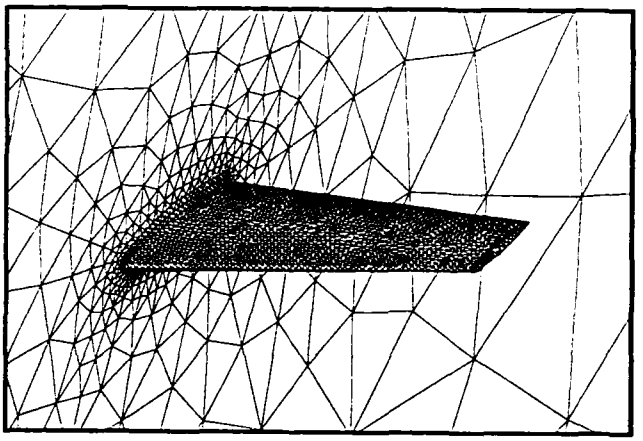


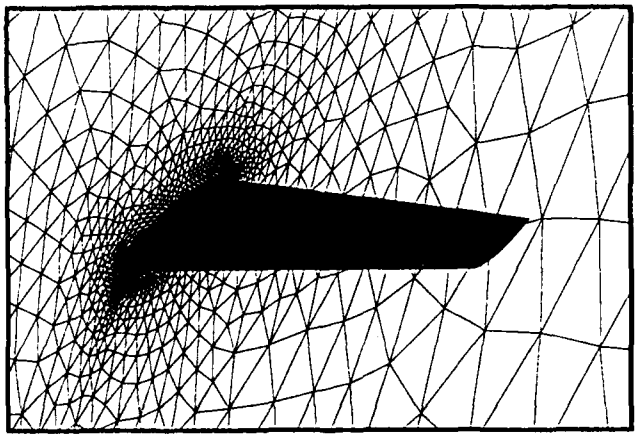
Figure 3: Data Access Pattern of 2800 Node Mesh After RCM Reordering



Mesh 1



Mesh 2



Mesh 3

Figure 4: Sequence of Global Coarse and Fine Meshes Employed for Computing Inviscid Transonic Flow over the ONERA M6 Wing. Mesh 1: 2,800 Nodes, 13.576 Tetrahedra, 2,004 Boundary Faces; Mesh 2: 9,428 Nodes, 47,504 Tetrahedra, 5,864 Boundary Faces; Mesh 3: 53,961 Nodes, 287,962 Tetrahedra, 23,108 Boundary Faces; Mesh 4: 357,900 Nodes, 2,000,034 Tetrahedra, 91,882 Boundary Faces (Not Shown).

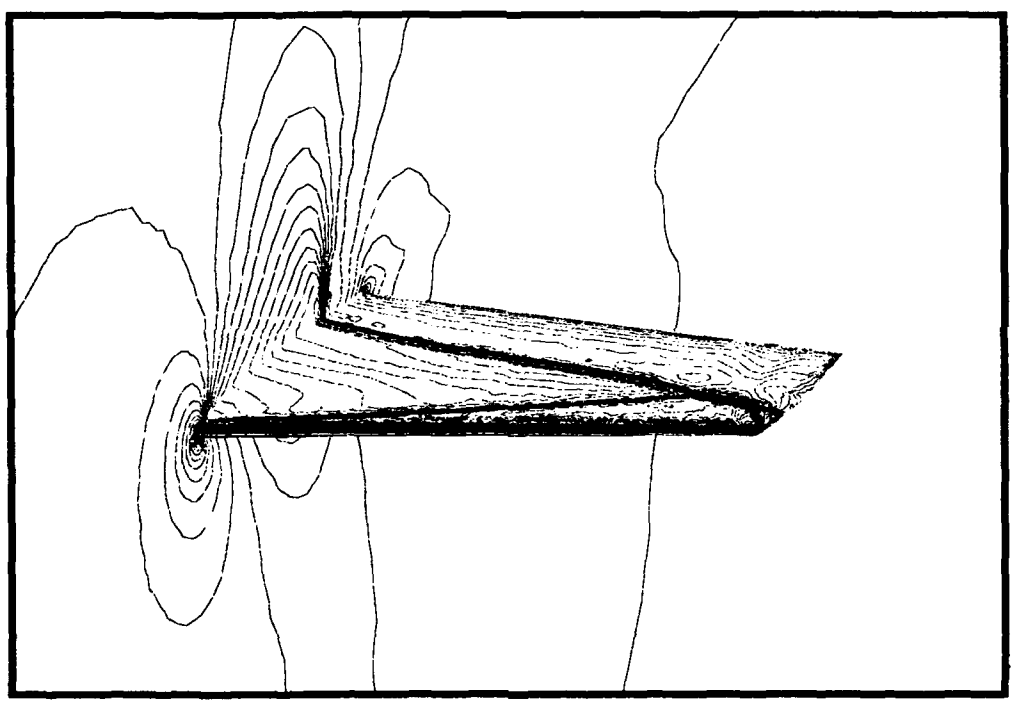


Figure 5: Computed Mach Contours on the Second Finest Mesh of the Multigrid

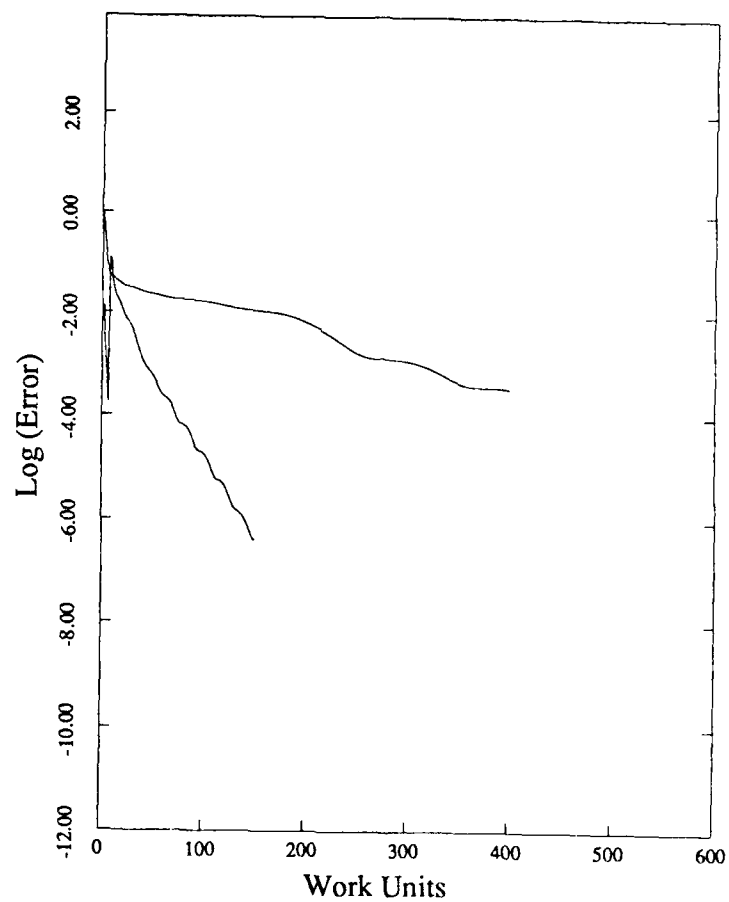


Figure 6: Comparison of the Multigrid Convergence Rate and the Single Grid Converge Rate on the Finest Grid of the Sequence about the ONERA M6 wing as Measured by the Average Density Residuals versus the Number of Work Units

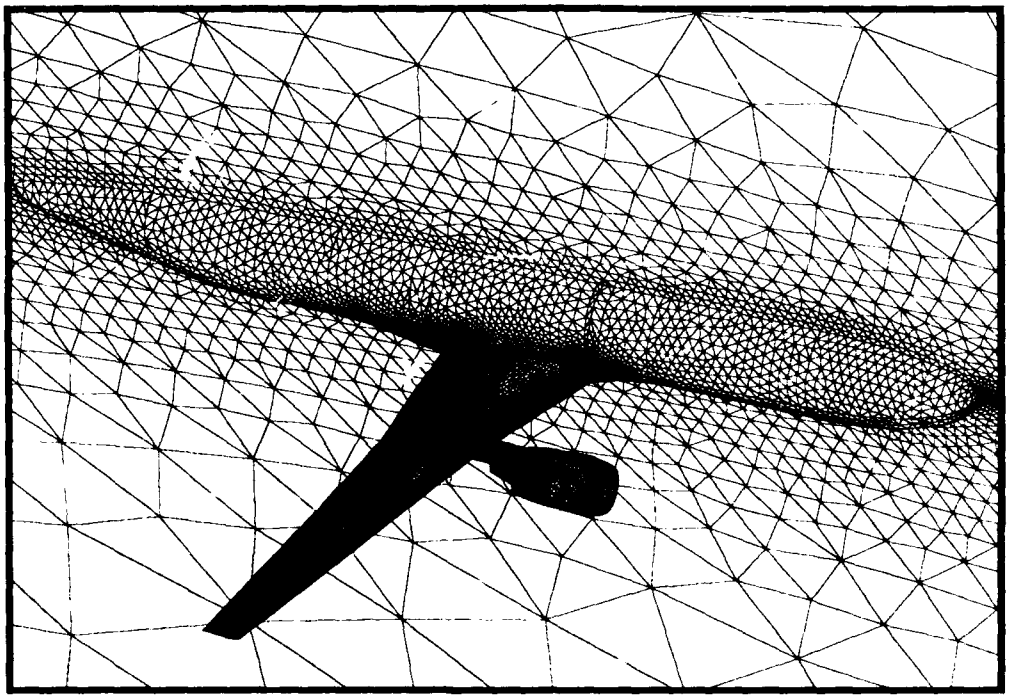


Figure 7: Coarse Unstructured Mesh about an Aircraft Configuration with Single Nacelle; Number of Points = 106,064, Number of Tetrahedra = 575,986 (Finesh Mesh Not Shown)

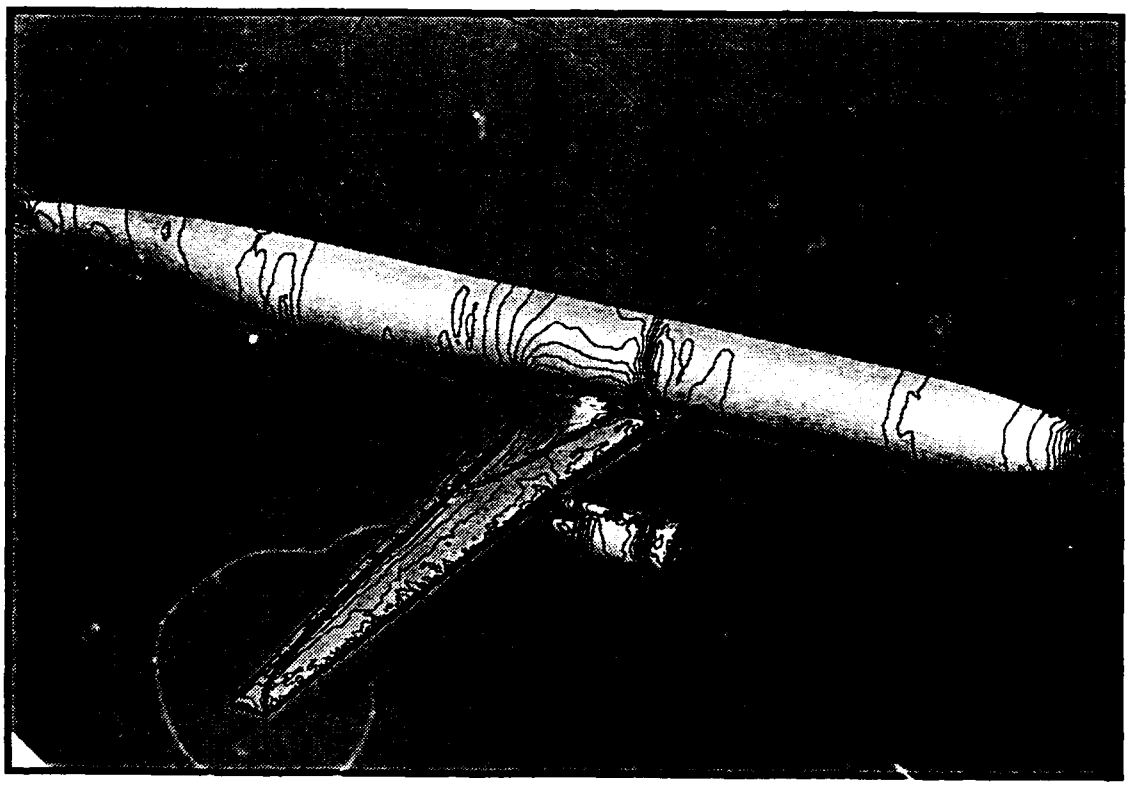


Figure 8: Mach Contours for Flow over Aircraft Configuration Computed on Fine Mesh of 804,056 Vertices and 4.5 million Tetrahedra (Mach = 0.768, Incid ence = 1.116 degrees)

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1992	Contractor Report

4. TITLE AND SUBTITLE
THE DESIGN AND IMPLEMENTATION OF A PARALLEL UNSTRUCTURED EULER SOLVER USING SOFTWARE PRIMITIVES

5. FUNDING NUMBERS
C NAS1-18605
WU 505-90-52-01

6. AUTHOR(S)
R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 92-12

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING MONITORING AGENCY REPORT NUMBER
NASA CR-189625
ICASE Report No. 92-12

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
Proceedings of the AIAA Reno Conference - 1992

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 02, 61

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This paper is concerned with the implementation of a three-dimensional unstructured-grid Euler-solver on massively parallel distributed-memory computer architectures. The goal is to minimize solution time by achieving high computational rates with a numerically efficient algorithm. An unstructured multigrid algorithm with an edge-based data-structure has been adopted, and a number of optimizations have been devised and implemented in order to accelerate the parallel computational rates. The implementation is carried out by creating a set of software tools, which provide an interface between the parallelization issues and the sequential code, while providing a basis for future automatic run-time compilation support. Large practical unstructured grid problems are solved on the Intel iPSC/860 hypercube and Intel Touchstone Delta machine. The quantitative effect of the various optimizations are demonstrated, and we show that the combined effect of these optimizations leads to roughly a factor of three performance improvement. The overall solution efficiency is compared with that obtained on the CRAY-YMP vector supercomputer.

14. SUBJECT TERMS	15. NUMBER OF PAGES
parallel processing; unstructured Euler-solver	26
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		